Describing Web Service Architectures through Design-by-Contract

Sea Ling[1], Iman Poernomo[1], and Heinz Schmidt[1]

School of Computer Science and Software Engineering,
Monash University, Caulfield East, Victoria, Australia 3145.
{sling,ihp,hws}@csse.monash.edu.au

Abstract. Architectural description languages (ADLs) are used to specify a high-level, compositional view of a software application, specifying how a system is to be composed from coarse-grain components. ADLs usually come equipped with a formal dynamic semantics, facilitating specification and analysis of distributed and event-based systems. In this paper, we describe the Radl [12], an ADL framework that provides both a process and a structural view of web service-based systems. We use Petri-net descriptions to give a dynamic view of business workflow for web service collaboration. We adapt the approach of [14] to define a form of design-by-contract [10] for configuring workflow architectures. This serves as a configuration-level means of constructing safer, more robust systems.

1 Introduction

Business-to-business (B2B) requires software solutions that are business-oriented, mission-critical, scalable and distributed. In meeting these demands, the software engineer can benefit from a formal design methodology. Different methodologies should be employed to design different aspects of a system. For example, UML [11] is useful to describe fine-grain, object-oriented design of individual components of a system.
In this paper, we outline an approach to specification and analysis of *architectural* aspects of middleware-based enterprise systems. These aspects consist of coarse-grain, structural design decisions that need to be made to facilitate quality requirement of the system. Architectural description languages (ADLs) are used to define and analyse architectural designs, as configurations of interoperating components. Generally, this is enabled through the use of a static syntax, for defining configurations of components, and a dynamic semantics, for analysing the behaviour of configurations. The formal nature of ADLs enables us to meet many of the needs of the enterprise at the architectural level. Architectural description can help us understand scalability and distribution issues to (partially) satisfy mission-critical demands.
The past few years has seen the emergence of web services as a B2B enabling technology [7]. In this paper, we will be concerned with workflow architectures that are implemented through configurations of web services.
We describe how our ADL, Radl [12], addresses B2B concerns, through the architectural description and analysis of web service based systems. We will be concerned with configurations of web services that implement a required business workflow. A workflow is a specific ordering of work activities across time and place, with a beginning, an end, and clearly defined inputs and outputs. Business workflows are both internal and external to autonomous business entities, defining their means of collaboration. An architecture for web service workflow deployment defines how the workflow's activities are to be organised: their location, hierarchical relationships, structural reuse and compositionality.
Our language can describe these various kinds of deployment, employing a uniform syntax and semantics for description and analysis. We define web service architectures through encapsulating workflows within the ADL. Our approach useful, because it provides us with a single framework in which to 1) design, understand and analyse the

overall business-logic of a system, and 2) investigate and specify possible combinations of web service-oriented middleware for a workflow deployment.
Our work also uses a Petri-net semantics. We organise hierarchies of interacting web service workflows, each implementing a reusable, encapsulated business process. For the purposes of modelling, we use an abstract view of a web service, similar that of traditional ADLs for component modelling. We specify web services in terms of both workflows that are needed and public views of workflow that are available for participation. We employ an object-oriented type inference system, defined over web service interface descriptions, to facilitate web service substitutability based on behavioural inheritance. Then, we provide a novel adaption of recent approaches to architectural design-by-contract [13, 10] to this context, defining when and how two workflow web services can safely interact with each other. These approaches give us a means of obtaining safer notions of interoperability between workflow architectures.
We proceed as follows. Section 2 provides some required preliminary definitions, summarizing how Petri nets may be used to model workflow. Section 3 defines our ADL for web service description. Section 4 provides a small case study. In section 5, we identify related work and provide concluding remarks.

2 Process description using Petri nets

Petri nets have been widely used for process definition, covering behaviours of sequential execution, non-determinism and/or concurrency. (For a detailed description of Petri nets, see, e.g., [5]).
It is natural to use Petri nets for business process or workflow specification [4]. A workflow is a set of coordinated tasks to fulfill a specific business purpose [16]. In a special class of Petri nets, called Workflow nets (WF-nets) [2], workflow concepts are modelled by Petri net elements: workflow *activities or tasks* are modelled by transitions (graphically depicted by rectangles), *conditions* (precedence relations) modelled by places (depicted by circles), flow directions modelled by directed arcs and *cases* modelled by tokens. Standard workflow building blocks, such as AND-split, AND-join, OR-split and OR-join, have been represented by various net structures. Petri nets are strictly more powerful than the standard workflow graphs, since a number of high-level net classes [8] extend standard Petri nets in ways permitting states to carry complex and structured cases (tokens), and allowing activities to be parametrized, without losing the analytical power of Petri nets.
A Petri net $N = (P, T, F)$ is a *Workflow net (WF-net)* if and only if N has two special places $i \in P$ called *source* (an entry point) and $o \in P$ called *sink* (an exit point), and by adding a transition t^* to N, the short-circuited net $(P, T \cup \{t^*\}, F \cup \{(o, t^*), (t^*, i)\})$ must be strongly connected [1]. We use a contractor example described in [3] to illustrate a WF-net process specification, as shown in Figure 1.
Starting from source `i`, the contractor sends an order to some external subcontractor for a particular product. The `send order` tasks generates three concurrent threads of execution (AND-split): cost statement thread (first task: `prepare cost statement`), customer bill thread (first task: `bill`) and specification thread (first task: `create specification`). The choice (OR-split) structure of the process is illustrated by the checking of customer bill. At state `p1`, after the customer bill is created, the contractor has to make sure it is `okay`. If not, the bill has to be revised in a loop structure. After all three threads have been executed successfully, they will merge through the final task `handle product` that will lead to the sink `o`.
The tasks and process flow specified in Figure 1 are of local interest to the contractor organization and they are *private*. However, some tasks need to communicate and interact with external organizations (e.g. a subcontractor). In [3], these tasks are exposed as *public* methods so that interactions with the subcontractor organization or any other organization is possible. The combination of workflows is an interorganizational workflow [3] comprising several local workflows interacting with each other. The notion of workflow correctness or soundness has also been defined and relates directly to the liveness and boundedness properties of the underlying Petri net graph.

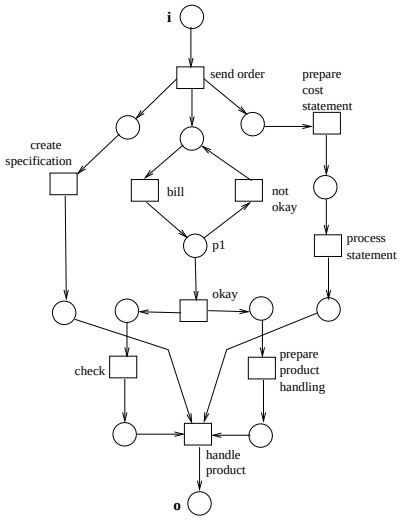

Fig. 1. A WF-net for the contractor.

Each local workflow can be viewed as an individual workflow web service. While [3] exposes the public methods as web service interface and hides the internals, in our work we shall distinguish between the *required* interface and the *provided* interface in addition to exposing the protocol of methods' invocation.
Workflows can be enhanced and altered by adding new tasks. To permit such workflow changes and still ensure correctness, a behavioural equivalence property in net structures has been formalized [3] based on branching bisimilarity [6]. Consequently, it is possible to define a notion of inheritance, called projection inheritance, which focuses on the dynamics rather than data and signatures of methods [3].

3 Architectural description

The Radl ADL [12] was originally described in [14], and has been extended by Ralf Reussner and two of the current authors in [13], to use finite state machines for behavioural and protocol descriptions and interoperability checks, specifically for component-based software.[1] The ADL has been used to model architectures for enterprise systems, by providing representations and semantics that meet enterprise concerns. In previous work, we have investigated .NET component-based architectures, and, in particular, configurations of business-centric .NET components. This paper now extends Radl to use Petri-nets for web-service description. First, we summarize basic aspects of Radl that are relevant to our current purpose. Our language has (semantically equivalent) textual and visual presentations. The former is useful for machine checking. The latter form is useful for human design and comprehension, and, for this reason, in this paper we define Radl in terms of the visual syntax.

3.1 Basic concepts

Architectural description languages (ADLs) present a compositional, web service-oriented view of software architecture. Our ADL achieves this through modelling configurations of

- *basic kens*, representing web services that are executable and available (for example, web services that have been located by a UDDI server),

[1] Radl is the product of continuing research conducted by the TrustME group at the DSTC and the School of Computer Science and Software Engineering, Monash University.

- *gates*, interfaces of kens, exposing public behavioural information about an associated ken,
- *connections* between required and provided *gates* of kens, the possible types of communication and interaction within a system, and
- *compound kens*, higher-level entities composed of interconnected kens, that may in turn be basic or compound. These specify how a new web service can be built from smaller web services.

Kens and gates are analogous to components and services respectively in Darwin, to web services and ports respectively in C2 and ACME, or to processes and ports in MetaH [9]. As interfaces of kens, gates provide public behavioural descriptions of how a ken can be used, or what a ken requires of other kens in order to function. These two uses of gates result respectively from two types of gates for kens: *provided* and *required* gates. In our case, our interface descriptions take the form of Petri-net workflows.

For the purposes of comprehending configurations, kens can be viewed at various levels of encapsulation. A *black-box* view of a ken does not provide details of how the ken implements the functionality described by the gate interfaces. A *white-box* view of a ken provides such detail. A white-box view of a basic ken consists of a private workflow description, with a mapping between tasks of this description, and the tasks exposed by the gates of the ken. A white-box view of a compound ken provides details of how a ken's functionality is built from configurations of subsidiary kens, each of which in turn is given as a white-box ken. A *grey-box* view of a compound ken provides some – but not necessarily all – details of how a ken's functionality is built from configurations of subsidiary kens. This view given subsidary kens as either white-box or black-box views.

An *architecture* is defined by a set of interconnected kens.

There are two forms of connections between gates: connection mappings and bindings. *Mappings* are hierarchical connections. They may occur between two required gates or two provided gates, where the first gate belongs to a ken that is contained within the second gate's ken. *Bindings* are connections between required and provided gates of kens. They are non-hierarchical, because the kens must at the same level of compositionality: for binding to take place, one ken cannot be contained within the other.

Gates possess a richer language than, say, ports of Darwin [9]:

- A gate is associated with a signature for type checking.
- Gates are instances of gate classes. Consequently, gates are analogous to interface objects, adaptors or wrappers in programming.
- Designers can protect existing functionality within gates, but also it permits the substitutions between existing kens in a configuration, subject to compatibility checks over gates.

These features enable us to provide web services specifications as kens. In particular, multiple interfaces exposed by a web service are simply modelled by a web service ken with multiple gates.

3.2 Protocol between gates of a ken

In contrast to most ADLs, we define a public protocol between provided an required gates. Visually, this is represented as a workflow transition between a tasks of separate gates. This provides a description of how tasks, accepted by a required gate, are subsequently sequenced with those tasks that are produced by the required gate. This description provides encapsulation: a black-box view of the ken does not review the details of how this sequencing arises.

The need for public description of interactions between provided and required interfaces does not arise for most ADLs, because these languages are not designed for the purpose of modelling workflow architectures. In these cases, interfaces can be presumed as purely asynchronous and independent of each other. In the case of workflow, this is generally not the case. In the case of the `Subcontractor` example, the

workflow between gates specifies that the processing of a specification is required to be completed prior to sending out a cost.
In this way, a ken is given a public workflow through its gates. This provides the architect with compositional constraints that lead to safer designs: whether or not kens may be connected (*compatibility*) and or may be substituted for one another (*substitutibiltiy*) within an architectural configuration. These notions can be formally understood through behavioural equivalence of gate Petri-nets.

3.3 Class-based organisation of architectures

Radl employs object-oriented mechanisms for its definitions, to give us a notion of correct substitutibility between workflow web services. A ken is an instance of a *ken class*, and a gate is an instance of a *gate class*. These kinds of classes may be specialized through object-oriented subclassing. This permits reuse of specifications.
We define an object-oriented type system for architectures in the following manner.

Definition 1 (Ken subtyping). *Let K and K' be two kens classes. Let $\bar{p}^K = [p_1^K, \ldots, p_i^K]$ and $\bar{p}^{K'} = [p_1^{K'}, \ldots, p_j^{K'}]$ denote the provided gates of K and K', respectively. Let $\bar{r}^K = [p_1^K, \ldots, p_i^K]$ and $\bar{r}^{K'} = [p_1^{K'}, \ldots, p_j^{K'}]$ denote the required gates of K and K', respectively. Let wf^K and $wf^{K'}$ denote the public workflow of K and K', respectively. We say K' is a subtype of K, writing $K' \leq K$, when*

$$\begin{aligned} wf^K &\sqsubseteq wf^{K'} \\ \bar{p}^K.wf &\sqsubseteq \bar{p}^{K'}.wf \\ \bar{r}^{K'}.wf &\sqsubseteq \bar{r}^K.wf \end{aligned}$$

These conditions correspond to saying that a ken K' can be substituted for a ken K when

1. The workflow of K is preserved by K' – K' may now involve new tasks, but still permits the original sequencing of tasks of K.
2. K' may provide larger workflows, involving more tasks, but it must at least expose the original functionalities of K.
3. K' can require less interaction with other workflows, but cannot require more than that needed by K.

3.4 Design-by-contract

The class-based organization forms a kind of design-by-contract [10], adapted to the architectural design level [13]. The requirement for safer connections between web services is satified by assigning interfaces to gates, because validity of connections between web services is determined by type checking.
Within the domain of programming language design, there are convincing arguments for adding further semantic annotations to web service interfaces, to make interface usage safer. Essentially, it is argued that an interface signature alone does not tell us *what* the interface methods are supposed to do, and that this information is necessary to use the interface safely.
This is difficult to achieve in most alternative ADLs. Most ADLs model a component's interface as a signature of required tasks and provided events. A behavioural specification tells us how these tasks and events are related to each other. These ADLs define interface elements to possess no signature or specification, so interoperability between components is understood as a simple binding between interfaces of web services. However, the task of determining whether interfaces *should* be connected is orthogonal to architectural design. This can be acceptable for defining small-scale architectures, where communication between web services is simple (such as Unix scripts which use pipes and filters). Unfortunately, for larger architectures, where complex information is being communicated, this situation leads to increased risk of design failure. This situtation is therefore not satisfactory for modelling large web service based systems.

4 Case study

In our case study, we use an example of an interorganizational workflow introduced by Aalst [3], involving two business partners: a contractor and a subcontractor. That is, the previous contractor workflow example in Figure 1 is expanded to include communications with a subcontractor. The contractor sends a product order to the subcontractor followed by a detailed product specification. The subcontractor then sends a cost statement to the contractor. Based on the product specification, the subcontractor manufactures the product and sends it to the contractor.

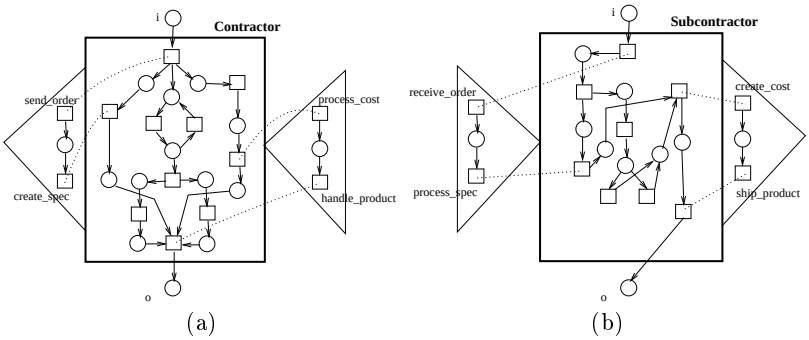

Fig. 2. (a) Contractor workflow. (b) Subcontractor workflow.

Figure 2 illustrates (a) the contractor workflow web service and (b) the subcontractor workflow web service. Note that the internal structure of Figure 2 corresponds to Figure 1. As in [3], a list of methods is offered to other business domains. We also specify the expected order of execution for these methods, using Petri nets. For example, `send_order` must occur before `create_spec`, and `process_cost` before `handle_product`. These methods corresponds to some tasks within the internal workflow description of the web service, as depicted by the dotted lines in the figures. The description, hidden from the external environment, is the actual workflow to be executed, called the private workflow in [3]. It consists of tasks which are only of local interest.

The contractor's provided gate provides services `process_cost` and `handle_product`, while the subcontractor's provided gate provides `receive_order` and `process_spec`. Workflow methods, associated with these services, expect some form of input data or trigger in order to be executed. The Petri net on the provided gate also outlines the acceptable protocol method execution sequence. Symmetrically, a required gate represents possible connections to other web services that need to perform the services provided. The Petri net on the required gate can be interpreted as an abstraction of the call sequence potentially generated during services provided.

The two web services can be connected through appropriate methods to form an interorganizational workflow as shown in Figure 3. For example, `send_order` sends an order to `receive_order` while `ship_product` ships the product to `handle_product`.

However, connecting the appropriate methods is insufficient to ensure correct workflow execution. The gates must also provide information detailing the relationship between the required net and the provided net of a web service. In fact, `send_order` and `create_spec` must precede `process_cost` and `handle_product` for the contractor web service, while `receive_order` and `process_spec` must be followed by `create_cost` and `ship_product` for the subcontractor web service. A deadlock will occur if the above conditions are not met. The workflow, only visible to the external world, details the order by which the interface methods should be executed. Adhering to the

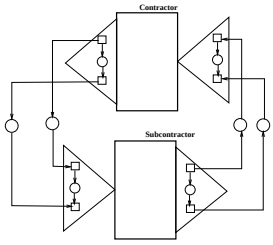

Fig. 3. The interorganizational workflow.

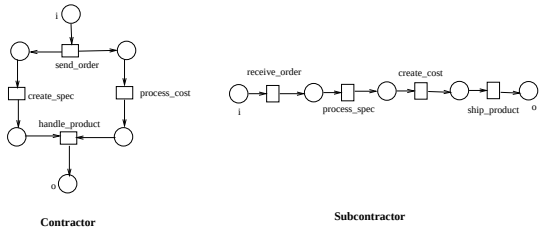

Fig. 4. Public workflows for interface methods

expected behaviour of the web services, the public workflows are shown in Figure 4. This information should be given to the web service users to ensure correct workflow operations.

5 Related work and conclusions

Most research on architectural description languages has been primarily concerned with architectures of concurrent, distributed, event-based software. To the best of our knowledge, this paper is the first attempt at using an ADL to describe web service architectures and to address B2B concerns. While addressing many concerns for enterprise design and analysis, this work has not explicitly focused on business-oriented concerns, nor, in particular, the problem of modelling enterprise workflow architectures. Most ADLs involve a dynamic semantics, useful for describing and understanding distribution and concurrency in architectures. However, such languages can be impractical for describing enterprise workflow.
The notable exception is [3], where an ADL with a Petri net semantics was developed, facilitating web service-based organisation and analysis of workflow. Petri nets have been used effectively for workflow description, and tools exist to intergrate Petri net descriptions into implementations in various workflow engines. However, that language is not as rich as traditional ADLs, where language web services are usually specified in terms of provided interfaces, functionality that the web service exposes as services to other web services, and required interfaces, services that the web service can use from other web services.
Object-oriented notions of behavioural subtyping have been investigated in [14]. Our adaptation of this notion to architectures is comparable to that of C2: both make use of of a behavioural equivalence partial order over web service interfaces.
Our conditions for subtyping are stronger than those of Aalst, because we do not only consider the overall public workflow of a web service, but also the individual workflows defining the provided and required gates.
Our approach emphasizes two issues that are important to enterprise workflow modelling. The first is modelling safer usages between web services by means of design-by-contract in our ADL. The second issue is provision of closer correspondence between dynamic semantics and workflow description in practice. Most ADLs involve

a dynamic semantics, useful for describing and understanding distribution and concurrency in architectures. However, such languages can be impractical for describing enterprise workflow. On contrast, Petri nets have a long history of use in describing workflow, and tools exist to integrate Petri net descriptions into implementations in various workflow engines.
Current state-of-the-art in modelling enterprise workflow is to use the BizTalk or Oracle Workflow notations and tools. Also, in the domain of web service workflow description, the WSDL has been proposed as a standard [7]. It is an open area of research to adapt our methods to for these other notations. This should be possible, because Petri nets have been shown to be a good formalism for formally representing many workflow notatins.

References

1. W. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997. 18th International Conference Proceedings*, pages 407–26, LNCS 1248, Springer-Verlag, 1997.
2. W. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W. van der Aalst. Inheritance of interorganizational workflows: How to agree to disagree without loosing control. Technical Report BETA Working Papers Serise, WP46, Eindhoven University of Technology, The Netherlands, 2000.
4. W. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques and Empirical Studies.* Springer-Verlag, 2000.
5. J. Desel and J. Esparza. *Free Choice Petri Nets.* Cambridge University Press, 1995.
6. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):550–600, 1996.
7. W3C Group. Web services activity site. Web site, W3C, 2002. Available at http://www.w3.org/2002/ws/.
8. K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets, Theory and Application.* Springer-Verlag, 1991.
9. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Janurary 2000.
10. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice/Hall, 1997.
11. OMG. OMG Unified Modeling Language Specification. Technical report, Object Management Group, 2000.
12. Iman Poernomo, Heinz Schmidt, and Ralf Reussner. The TrustME language site. Web site, DSTC, 2001. Available at http://www.csse.monash.edu.au/dsse/trustme.
13. Ralf Reussner, Iman Poernomo, and Heinz Schmidt. Using the trustme tool suite for automatic component protocol adaptation. In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 854–863. Springer-Verlag, 2002.
14. Heinz Schmidt. Compatibility of interoperable objects. In *Information Systems Interoperability*, pages 143–199. Research Studies Press, Taunton, Somerset, England, 1998.
15. Heinz Schmidt and Ralf Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, Monash University, School of Computer Science and Software Engineering, 2000.
16. Workflow Management Coalition. Workflow management coalition terminology and glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussals, February 1999.